pioman: a pthread-based Multithreaded Communication Engine

Alexandre Denis

pioman: a pthread-based Multithreaded Communication Engine

Alexandre DENIS
Inria Bordeaux – Sud-Ouest, France
E-mail: Alexandre.Denis@inria.fr

***Abstract*—Recent cluster architectures include dozens of cores per node, with all cores sharing the network resources. To program such architectures, hybrid models mixing MPI+threads, and in particular MPI+OpenMP are gaining popularity. This imposes new requirements on communication libraries, such as the need for `MPI_THREAD_MULTIPLE` level of multi-threading support. Moreover, the high number of cores brings new opportunities to parallelize communication libraries, so as to have proper background progression of communication and communication/computation overlap. In this paper, we present `pioman`, a generic framework to be used by MPI implementations, that brings seamless asynchronous progression of communication by opportunistically using available cores. It uses system threads and thus is composable with any runtime system used for multithreading. Through various benchmarks, we demonstrate that our `pioman`-based MPI implementation exhibits very good properties regarding overlap, progression, and multithreading, and outperforms state-of-art MPI implementations.**

I. INTRODUCTION

The advances in processor architecture has brought computing into the multicore era. A typical cluster node nowadays is comprised of tens of cores. Manycore processors bring us one step forward with hundreds of threads. With such number of cores per node, the approach to program clusters has to evolve. The "pure MPI" model, with one process per core, will not scale forever. New hybrid programming models emerge, that mix MPI and threads, to better exploit resources.

However, mixing threads and communication is not straightforward [1]. Supporting `MPI_THREAD_MULTIPLE` multithreading level requires careful design. Even though it brings constraints, multithreading may be seen as an opportunity to parallelize the communication library itself for packing, checksums, or multi-rail for example [2], [3]. Parallelism in the communication library is useful also for asynchronous progression of communication, which allows efficient overlap of communication and computation. This property allows applications to hide the cost of communications [4], [5], [6].

In this paper, we present `pioman`, a generic framework to be used by MPI implementations, that brings seamless asynchronous progression of communication by opportunistically using available cores. It uses system threads and thus is composable with any runtime system used for multithreading.

The remaining of this paper is composed as follows. Section II analyzes related works. Section III presents our approach for a communication progression engine supporting asynchronous progression and multithreading. Section IV analyzes concurrency and proposes a design that alleviates contention. Section V presents a performance evaluation of our implementation. Finally, Section VI draws a conclusion of this study and shows directions for further work.

II. RELATED WORKS

With the advent of multicores, multithreaded communications is a hot topic. OpenMPI [7] has some mechanisms to make communication asynchronously progress in its drivers. It successfully overlaps communication and computation on the sender side. It supports `MPI_THREAD_MULTIPLE` multithreading with its TCP driver, not *InfiniBand*. MVAPICH 2 [8] has reportedly mechanisms to overlap the *rendez-vous* protocol with computation, taking benefit of the *InfiniBand* hardware being able to autonomously perform an RDMA transfer. Rashti *et al.*[9] have proposed a similar mechanism based on the properties of RDMA. MVAPICH 2 supports `MPI_THREAD_MULTIPLE` on *InfiniBand* though it is not built by default and must be explicitly configured. MT-MPI [3] is based on MPICH and integrates closely with an OpenMP runtime to opportunistically exploit idle cores to make communication progress; it is specific to Xeon Phi and to the given OpenMP runtime. Wittman *et al.*[10] have proposed a framework that makes any MPI implementation able to asynchronously progress, given that it supports `MPI_THREAD_MULTIPLE`, which is very restrictive since few MPI implementations support this feature. Hoefler *et al.*[11] have studied the benefit of using multithreading in communication engines, and have concluded that a thread dedicated to communication progress is beneficial in some cases, but not when it competes with computation. More advanced mechanisms are needed, which is quite in accordance with our proposal. Our own previous work [12], [13] on `pioman` proposed mechanisms for asynchronous progression. However, it was bound to the *Marcel* thread scheduler, which prevented it to be composed with other runtime systems.

III. A MULTITHREADED COMMUNICATION ENGINE

In this Section, we present our approach for a communication progression engine supporting asynchronous progression and multithreading.

A. Tasklet-based communication progression

Parallelizing network communication processing is needed for asynchronous progression and for multithreaded application having their communication actually progress in parallel.

Such kind of mechanism has been well known and largely used in Linux kernel for several years. Interrupt handling is split in two parts: *top half*, the real interrupt handler executed with masked interrupts, and should be as short as possible; the *bottom half*, deferred work scheduled by the top half to be done later, doing the real work in a safer context, with unmasked interrupts. Originally, bottom halves (BH) were designed to improve reactivity and responsiveness, by moving non-urgent work outside of the interrupt handler. It was shown [14] that Linux kernel 2.2.x series suffered from bad communication performance in multithreaded contexts, and that the communication stack, in particular its bottom half, needed to be multithreaded for such loads. The bottom half was transformed into a collection of mechanisms (tasklets, softirqs, work queues, timers) [15] now collectively known as *bottom half* since kernel 2.3.x series.

The *tasklets* are small tasks to be executed asynchronously at some time later. The kernel ensures some guarantees on the context in which tasklets are executed, such as guarantee on concurrent operations (a tasklet is strictly serialized with itself, it cannot be executed on two cores at the same time), on deadline (scheduled no later than next timer tick), on CPU placement (same CPU as the one from which the tasklet was scheduled). The tasklets are a useful tool to parallelize the communication stack: they opportunistically utilize available resources, and asynchronously make communication progress independently of the application execution flow. The guarantees on their execution context make them easier to use than a random task scheduler.

B. `pioman`: tasklets in user-space with `pthread`

We propose to take benefit from the lessons learned from the kernel experience and to write a communication library with basic operations expressed as little tasks that may be executed by a progression engine that opportunistically utilize available resources. Then, making communication progress asynchronously and in parallel is only a matter of scheduling tasks.

We proposed [13] a first implementation of a task scheduling system for communication libraries called `pioman`. This version was closely integrated with a specific thread scheduler, which made its implementation easier with direct hooks in the scheduler. However, it restricts portability and usable applications.

In this paper, we propose a full rewrite of `pioman` using system threads (`pthread`), so as to be compatible with multithreaded applications, whatever the multithreading runtime (raw `pthread`, OpenMP, TBB, PGAS, etc.) or the compiler (e.g. Intel OpenMP v.s. GNU OpenMP). Its light tasks are called *ltasks*, which are inspired from tasklets but not completely mimics their behavior since user-space and kernel-space are different contexts with different requirements. These *ltasks* need to be executed at the following *polling points*:

- **idle–** for an opportunistic resource usage, when the application leave some cores available, we execute *ltasks* on resources detected as *idle*. For parallel progression, we should execute multiple *ltasks* if multiple cores are idle. However, we must take care that it does not cause contention or too much overhead.
- **timer–** to ensure guaranteed communication progression even when no resource are idle, and to have bounds on reactivity time, we need to have *ltask* execution triggered by timers.
- **explicit polling–** for a progression at least as efficient as the no-*ltask* flavor, we need to execute *ltasks* on explicit polling points.

Using system `pthread`, we do not have direct access to hooks that would be triggered upon idle or at each scheduler time slice. However, we can implement mechanisms that will wake-up at these specific times without hooks. We propose the following scheme.

Polling on idle is performed by one or several threads running with low priority or scheduling class. Where available, scheduling policy `SCHED_IDLE` is used, which has precisely the semantics we need; if not available, we use the lower available priority in policy `SCHED_OTHER`. The number and placement of idle threads depends on machine topology as explained in Section IV-A. They use throttling as described in Section IV-C.

For poll on timer, multiple solutions may be envisioned. The most precise periodic execution is the POSIX timer interface. However, it triggers execution of a signal handler, which is a restricted context where it is not possible to use locking nor any standard function that may use locking internally (`malloc`, IB verbs, etc.). Calling networking primitives from there causes random deadlocks because of this. It is similar to restrictions in interrupt context, which led to bottom halves in the first place. A better solution consists in creating a regular thread that sleeps and wakes up at regular interval. Policy `SCHED_FIFO` is a good candidate, but requires root privileges and may lead to deadlocks when sharing locks with threads with regular priority. In the general case, we use regular thread with high priority if available, else normal priority. Since this thread spends most of its time sleeping and consumes little actual CPU time, the kernel scheduler detects it as I/O bound and gives it a priority *bonus* when computing dynamic priorities. As a result, even with normal static priority, the timer thread does not starve in the presence of overload of computing threads. The period is not guaranteed, though.

Explicit polling is still required for good performance. When the application does a busy wait, it is beneficial to execute *ltasks* directly from there rather than waiting that timer or idle threads make communication progress. Even if the system is idle, executing *ltasks* from the context of the application thread saves context switches, which saves hundreds of nanoseconds [1] on the critical path. Therefore `pioman` exposes in its API a function for busy waiting that executes all *ltasks* once.

Mechanisms exposed here are able to schedule tasklet-like small tasks from user space at relevant polling points, using only POSIX standard API, fairly widespread among systems,

degrade nicely if wanted features are missing, and agnostic about runtime system used for multithreading.

C. Implementation and integration: `pioman` and NewMadeleine

We implemented these mechanisms in the `pioman` I/O manager, which is released as an *open source* standalone library available for download. It may be used by any communication library to make its communication progress asynchronously. For the sake of simplicity, we have integrated it with our own *NewMadeleine* communication library and evaluate it through MadMPI, a minimalist MPI interface atop *NewMadeleine*. However `pioman` is in no way bound to *NewMadeleine* and any other MPI implementation may be ported on it. `pioman` used to be bound to the *Marcel* multithreading library. The new version described here is a complete rewrite available in three flavors: `pthread`, legacy *Marcel*, and *nothread* which implements only explicit polling, roughly equivalent to what most MPI implementations do internally. In this paper, we focus on the `pthread` flavor.

Integration of `pioman` in a communication library involves chopping the parts that needs to run asynchronously and execute them from *ltasks* rather than directly. In the case of *NewMadeleine*, code executed in *ltasks* is: driver operations (send/recv, both post and poll); message submission (applying optimizing strategy on packet flow). To synchronize operations between *ltasks* and other code, high-level synchronization objects are provided. They allow *ltask* code to set a status lock-free, and other code to test status lock-free. When the user explicitly waits for an operation to complete, we use a *fixed-spin* strategy: first, we perform a busy wait, with explicit polling in a busy loop; after a given timeout, code switches to passive waiting on a semaphore, and any status change from an *ltask* will trigger the semaphore to wake up user code.

IV. Managing concurrency without contention

In this Section, we study the contention on `pioman` structures accesses and we propose mechanisms to avoid it. Indeed, the core structure is the *ltasks* queue, shared between polling entities (idle and timer threads, explicit polling), and submitters. Operations on the queue (enqueue, dequeue, traversal) are performed at very high frequency from multiple entities, which causes contention if we do not take care. We propose two classes of mechanisms: improve locality to reduce the impact on performance of contention; optimize the locking scheme to reduce contention.

A. Locality

Given the increasing topology complexity of nowadays machines, topology has to be taken into account to reach high performance. If we omit to do so, pathological behavior arises, such as *cache ping-pong*: two (or more) threads work on the same data set —the *ltasks* queue in our case— but do not share any cache or are on different CPU sockets; then the data comes back and forth from one CPU to the other, processors eventually spend more time waiting for data and in the cache coherency protocol than to perform actual work, and efficiency collapses.

A known [16][13] way to manage *runqueues* on hierarchical architectures consists in hierarchical queues: a local queue is attached to each node of the topology, forming a tree of *ltasks* queues. Then CPU or cores mostly work locally in their local queue, avoiding concurrent access to shared queues.

We use `hwloc` [17] to discover the internal machine topology with all its levels (machine, sockets, caches, cores), prune unnecessary levels (without siblings, e.g. no need for a *socket* level on a single-socket machine), and attach a local *ltasks* queue to each remaining entity. Tasks are submitted in the queue where it most makes sense regarding locality of memory or device: driver *ltasks* are enqueued in the queue of the entity where the device is actually attached [18], usually at *socket*-level; *NewMadeleine* message submission *ltask* is submitted to the queue with the same scope as the application thread binding — or the full machine if user did not bind threads.

For polling, *ltasks* are dequeued and executed from the most local queue, then queues from parents are recursively dequeued up to the root of the tree attached to the full machine. A full execution is thus run bottom-up on a full branch from a leaf to the root, without interaction with the siblings, and eliminates contention with siblings.

A naive tree traversal would not however eliminate contention in the upper levels. To reduce contention near the root, we perform the recursive polling on the parent queue with a frequency divided by the number of siblings, taking into account that multiple children object will contribute to the polling on their parent. With such a strategy, for example a polling round originating from a core on a 32-core machine will actually go up to the machine-wide queue only once out of 32 times.

Empirically, *idle threads* per socket is a good compromise, with enough parallelism to make communication progress even in case of multi-rail, but without imposing too much pressure contention-wise. For the *timer thread*, as long as its period is several milliseconds, in the range of a time-slice, it does not seem relevant to worry about locality since it causes very little contention. A single timer thread traversing the full tree is a good-enough solution.

This approach assumes that at any time, we can determine the location of the current thread in `pioman`, which is a challenging issue. This information is not available cheaply when using `pthread`. The `hwloc` primitive returning the current location actually asks the OS. It involves multiple system calls which costs several microseconds. Therefore it is not acceptable to call this function every time we execute *ltasks*. Our solution consists in using a *cache* of the recent observed location, stored as thread-specific data using *thread local storage*. We update the actual location at a period of some hundreds of milliseconds to amortize the cost of these system calls. It works fairly well since most application bind their threads, we bind our idle threads, and location does not matter in the timer thread that traverse the full tree. Even if

application threads are not bound to cores, the kernel moves them not so often anyway.

B. Locking schemes

Another type of contention on queues is concurrent *ltask* enqueue by the communication library from an application thread, and *ltask* execution by another thread (idle, timer, or explicit from another thread doing busy wait). The contention is especially important since *ltask* execution may be very short — e.g. polling on InfiniBand takes a few CPU cycles — so that *ltasks* are enqueued and dequeued at high frequency. The contention materializes differently depending on the locking scheme used to protect concurrent accesses to the queues. We consider two main families of locking schemes to access the *ltask* queues:

- **lock-based–** a lock (mutex or spinlock) is held while a thread enqueues a new *ltask* or while a thread dequeues or execute an *ltask*. With this scheme, when a thread enqueues an *ltask*, it is in direct competition with threads dispatching *ltasks*. They are competing to acquire the same lock.
- **lock-free–** the queues are implemented as lock-free [19] FIFO in circular arrays, using atomic operations. One enqueue and one dequeue may take place concurrently at both ends of the queue without causing contention (padding may be required to get *head* and *tail* in different cache lines, though). However, lock-free does not mean spin-free. Multiple enqueues or multiple dequeues collide and may spin on a *compare-and-swap*, which is very similar to spinning caused by contended spinlocks. Since possible operations on lock-free FIFO are limited (enqueue/dequeue only, no traversal is possible atomically), dispatching *ltasks* means dequeue, execute, and enqueue again. Eventually application threads and polling threads are still in direct competition to enqueue *ltasks*.

Preliminary implementation of both schemes confirm intuition and exhibit bad performance, even when using classical tricks to reduce contention (ticket lock, CLH lock, spinning with exponential back-off). Mechanisms used to ensure locality described in previous Section avoid contention between polling threads, but not between polling thread and application thread, since the application thread enqueuing the *ltask* is not necessary on the same CPU as the driver. Moreover, locality is no *magic bullet* and does not prevent threads to locally compete to acquire a spinlock or to commit a *compare-and-swap*.

To solve the issue of contention between enqueues and polling, we propose an alternate locking scheme that combines the best of both worlds, lock-based for polling which enables traversal, but lock-free for submission, which eliminates competition between polling and enqueue:

- **submission queues–** we attach a companion queue dedicated to submission to each *ltask* queue. The submission queue is lock-free; the main queue has a spinlock. Tasks from the submission queues are dequeued by polling threads before an *ltask* execution round, and enqueued in the main queue once the spinlock is already held.

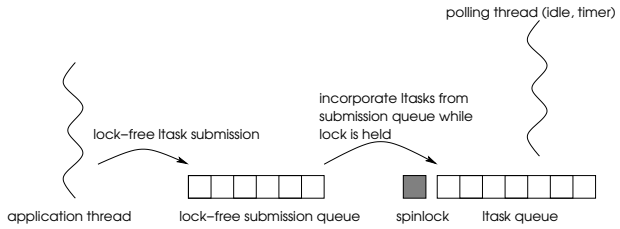

Fig. 1. Submission queues

This solution is depicted in Figure 1. It is loosely inspired from the two-part approach in RCU [20] (readers and writer use separate structures, changes from writer are incorporated later when there is no reader), but the original RCU is impractical here since it introduces too much delay to propagate changes, in the order of magnitude of milliseconds where we need submicrosecond to make it usable on the communication critical path. Our solution exhibits the following interesting properties:

- *ltasks* submission by the application is lock-free. The only possible competition is between enqueues from the application, which makes it almost spin-free in the general case. Submission does not touch the main queue nor its lock.
- locking on the main queue is used as a *mask*. When a polling thread tries to acquire the lock, if the lock is busy, it just skips the queue. Indeed, a busy lock means that another thread is already executing *ltasks* from the queue, so we can skip it safely. This locking mechanism is spin-free.
- when a polling thread incorporates submitted *ltasks* from submission queue into the main *ltask* queue, it holds the lock thus it is the single reader on the submission queue, and does not prevent application threads to submit *ltasks* concurrently in the submission queue thanks to the properties of lock-free queues.

The only interactions between threads that could introduce a performance penalty are: when incorporating *ltasks* from the submission queues, polling threads *read* head and tail, which are written by application threads; when trying to acquire a busy lock, a polling thread *reads* its value which was written by another thread. Interactions are limited to reading values written by other threads in some cases. Spinning on locks or atomics, and shared variable for writing are avoided. Contention is actually mitigated.

C. Throttling and tuning

Executing *ltasks* in busy loop on idle has an impact on computing and communications. Even though our locking scheme alleviates contention on *ltask* queues, busy loop – not `pioman` itself but actual network polling invoked from *ltasks* – imposes pressure on caches, buses, and TLB. To mitigate the impact of continuous polling, we implemented *throttling* in the idle thread loop. Between each *ltask* execution round, idle threads sleep for a given number of microseconds, or invoke `sched_yield` if period is set to 0.

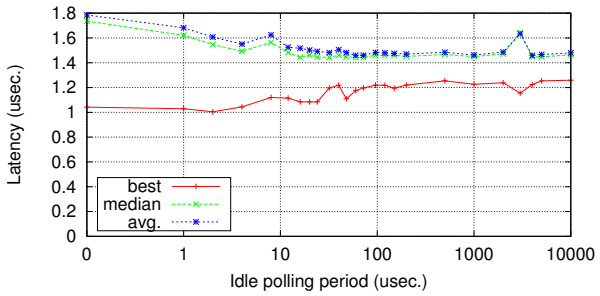

Fig. 2. 1-byte network latency against idle polling period, on host `william`. Best, median, and average latency for 10 000 000 round-trips.

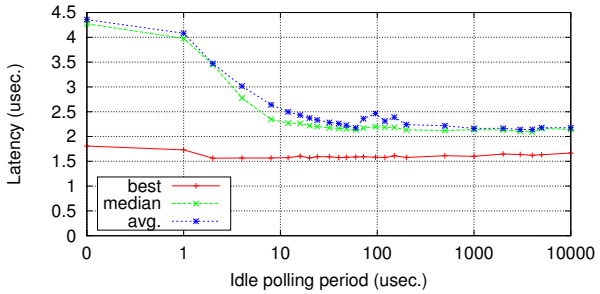

Fig. 3. 1-byte network latency against idle polling period, on host `joe`. Best, median, and average latency for 10 000 000 round-trips.

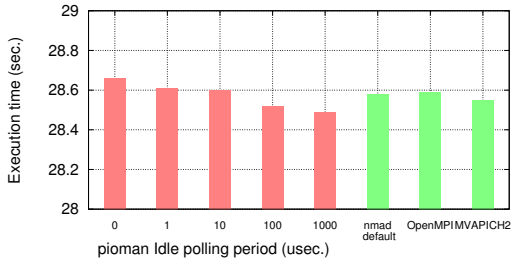

Fig. 4. ep.B.2 benchmark results on host `william`, median of 10 runs — Notice time origin is not 0, variations are less than 0.5 %.

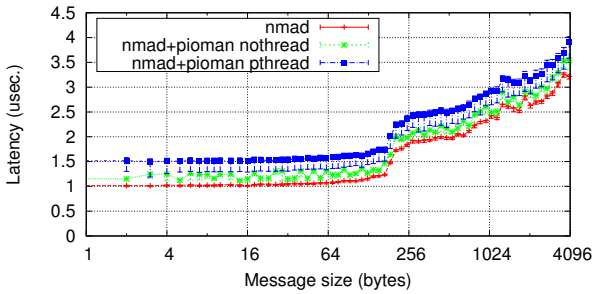

Fig. 5. Latency comparison between *NewMadeleine* flavors (error bars with min/median/average).

To tune the polling period, we have conducted some benchmarks with period ranging from 0 to 10000 μs (which is roughly equivalent to no polling on idle), and evaluated the impact on latency and computing performance. Figures 2 and 3 show latency results on two very different machines: `joe`, an old Xeon X5460 @3.16GHz, 4-core, single socket, equipped with IB ConnectX DDR; and `william`, more recent dual Xeon E5-2650 @2.00GHz, 16 cores (32 threads SMT), dual socket, NUMA memory, equipped with IB ConnectX3 FDR. Graphs depicts pingpong results for 10 millions round-trips with best latency, median, and average. The gap between best and median gives an estimation of dispersion. The gap between median and average gives an estimation of the length of the long tail. We observe that varying idle period has low impact on the best latency. Low period makes median and average higher, because of high dispersion of values; high frequency background polling causes jitter. The average remains close to the median, which shows that extreme values do not happen often. Any polling period beyond 10 μs has a negligible impact on latency, with overhead similar to what we get for the largest period. This period still ensures reasonable background polling frequency.

To evaluate the impact of idle polling on computation, we measure the performance of NAS benchmark EP, which is mostly computation with very little communication. Results are depicted on Figure 4. We observe that period from 0 to 1000 μs give a maximum difference between results less than 0.5 %. The default period gives performance in par with OpenMPI and MVAPICH 2. Finally, we studied the impact of the period in the *timer thread*. With a period from 1 ms to 100 ms, the latency results and NAS EP computation results are indistinguishable from each other. We conclude that *system noise* caused by `pioman` is unnoticeable.

V. Evaluation

In this Section, we evaluate the overhead of `pioman`-enabled *NewMadeleine* compared to non-threaded version, and evaluate progression and multithreading related aspects, and compare it against OpenMPI and MVAPICH 2. We used OpenMPI 1.7.4 built with `-enable-mpi-thread-multiple` and MVAPICH 2 2.0b built with `-enable-threads=runtime`. Everything was built from source using `gcc` 4.7.2. All benchmarks were performed on InfiniBand, on cluster `william`, except NAS benchmarks on cluster `graphene`, composed of ConnectX DDR (MT26418) cards on quad-core nodes equipped with Intel Xeon X3440. If not stated otherwise, plotted values are *median* of several thousands of round-trips.

A. Overhead evaluation

As a first part of the evaluation, before we evaluate the benefits, we evaluate the overhead caused by `pioman`. Overhead caused by background polling on computation (through context switches, TLB, or cache pollution) has already been evaluated in the previous section. Figure 5 presents latency comparison between three flavors of *NewMadeleine*: original flavor without `pioman`, with only integrated explicit polling;

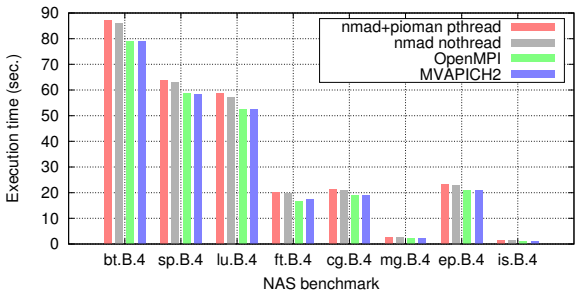

Fig. 6. NAS benchmarks.

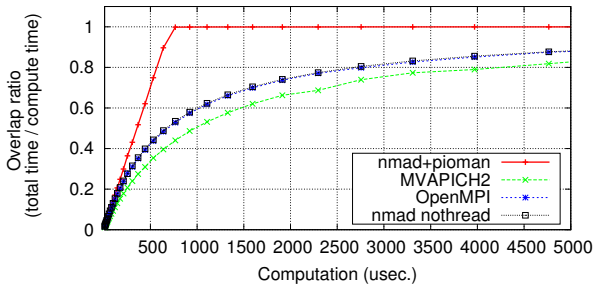

Fig. 7. Communication/computation overlap ratio, computation on both sides, 4 MB message.

`pioman`-enabled *NewMadeleine* without threads (and thus no background progression); `pioman`-enabled *NewMadeleine* with `pthread`. We observe that without threads, `pioman` introduces an overhead of roughly $100\,ns$. It is explained by the cost of enqueuing and dequeuing *ltasks* using lock-free atomic operations. When adding multithreading, and thus background polling, the best observed latency is roughly the same as without threads. However, the median is increased by another $100 \sim 200\,ns$. This overhead, although not completely negligible, is believed to be bearable.

As synthetic benchmark, we have run NAS Parallel Benchmark 3.3. Since NAS benchmarks do not overlap computation and communication, and since even the OpenMP+MPI MZ benchmark do not use MPI in the parallel sections, no gain is expected here. Results are depicted in Figure 6. `pioman`-enabled *NewMadeleine* performs equal or with an overhead below 2 % compared to non-threaded *NewMadeleine*. MVAPICH 2 and OpenMPI perform better on this benchmark, which is explained since MadMPI is only a proof-of-concept MPI implementation, with non-optimized collective operations. Overhead caused by multithreading support and progression is low.

B. Progression Benchmarks

To evaluate asynchronous progression of communication, that actually allows overlap of communication and computation, the usual metric [21] is the *overlap ratio*. It is defined as $overlap = T(computation + communication)/T(computation)$ with $T(computation)$ the computation time, and $T(computation + communication)$ the time of overlapped computation and communication. The overlap ratio reaches 1 for perfect overlap as soon as $T(computation) = T(communication)$ but anyway always converges towards 1 asymptotically for large computation time.

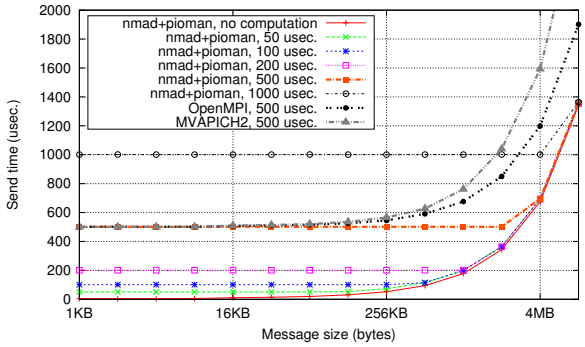

Fig. 9. Send time, computation on both sides.

Figure 7 reports overlap ratio with computation on both sender and receiver side, for various MPI implementations, for a 4 MB message and as a function of computation time. We observe that `pioman`-enabled *NewMadeleine* perfectly overlaps computation and communication, with an overlap ratio that reaches 1 for a computation time equal to the communication time. Other MPI implementations OpenMPI, MVAPICH 2, and the *nothread* flavor of *NewMadeleine* do not succeed to overlap computation and communication in this case. Figure 8 details the overlap ratio with computation only on sender side (graph on left), and on receiver side (right). We observe that OpenMPI actually overlaps computation and communication when computation is on sender side only.

Since we believe overlap ratio graphs are hard to read, we propose another metric. Figure 9 reports the *send time* with overlap both on sender and receiver side, with various computation time values, as a function of message size. The graph for a perfect overlap is expected to be $T(computation + communication) = max(T(computation), T(communication))$ and no overlap should be $T(computation + communication) = T(computation) + T(communication)$. We observe that `pioman`-enabled *NewMadeleine* perfectly overlaps computation and communication for computation time as low as $100\,\mu s$, and other MPI implementations have no overlap in this case. For clarity, OpenMPI and MVAPICH 2 are only graphed for $500\,\mu s$ computation, but results are similar with other computation time values.

Both benchmarks, overlap ratio and send time, show that `pioman` is actually able to successfully make communication progress in background, while MVAPICH 2 doesn't and OpenMPI only in some cases.

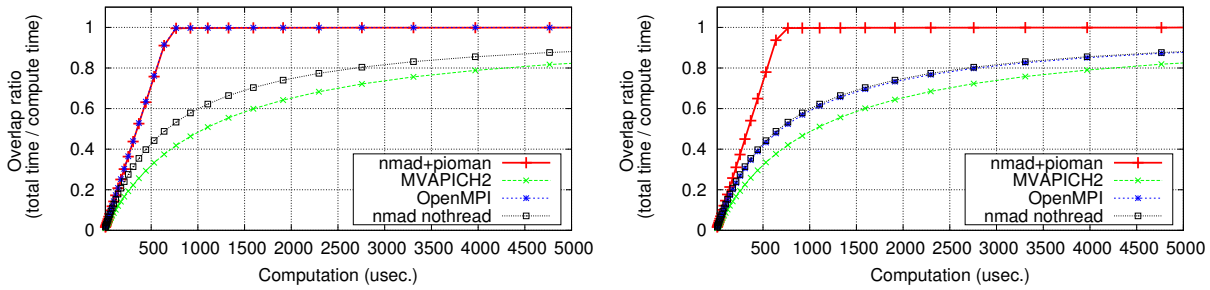

Fig. 8. Communication/computation overlap ratio, send-side only (left); receive-side only (right).

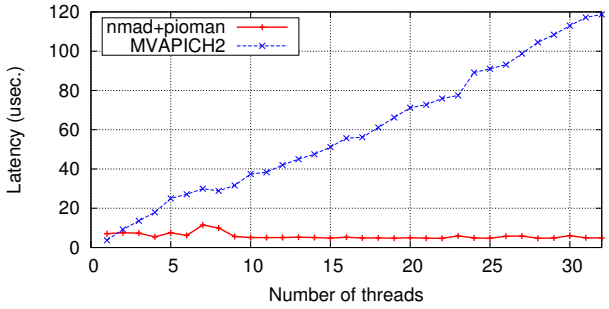

Fig. 10. 1-to-N: 1-byte latency for 1 sending thread, N receiving threads (median latency).

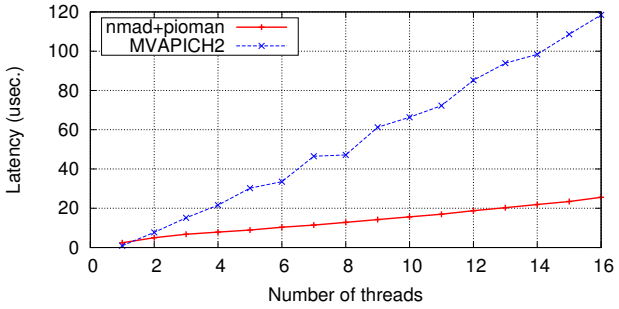

Fig. 11. N-to-N: 1-byte latency for N sending threads, N receiving threads (median latency).

C. Multithreaded Benchmarks

To evaluate multithreaded performance, we have considered three different benchmarks: 1-to-N, one sender thread, N receiver threads, this is `osu_latency_mt` from OSU MPI Benchmark [22]; N-to-N, N threads send data to N receiver threads, this is `latency_th` from Argonne thread test [23]; N-load, one sender thread, one receiver thread, N computing threads on both sides, from our own benchmarks. Since OpenMPI does not support `MPI_THREAD_MULTIPLE` on InfiniBand, it was not considered in 1-to-N and N-to-N benchmarks.

The 1-to-N benchmark is a 1-byte pingpong with one thread in a process and N thread in the peer process. Results are depicted in Figure 10. We observe that for `pioman`-enabled *NewMadeleine*, latency is roughly constant whatever the number of receiving threads, while for MVAPICH 2, latency is linear with the number of threads. In MVAPICH 2 all threads access the network board directly and compete with each other. In `pioman`, network access is performed in one *ltask*, while all application threads are waiting; then once matching is done, it wakes up the application thread through a semaphore – and the OS will actually wake up the *right thread* directly. This behavior does not depend on the number of threads waiting for a message and is thus in constant time on Linux with a $O(1)$ scheduler.

The N-to-N benchmark is a 1-byte pingpong between N thread in parallel. Results are depicted in Figure 11. We

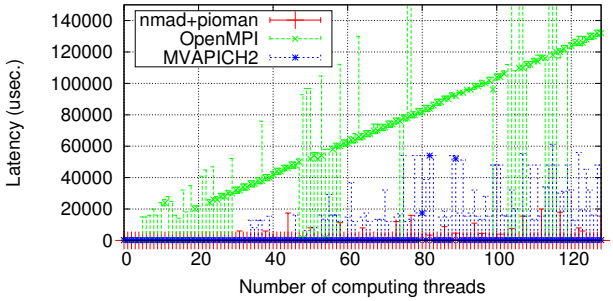

Fig. 12. N threads load: 1 MB latency for 1 sending thread, 1 receiving threads, N computing threads on both sides (error bars with min/max/median).

observe that for both studied implementations, the latency is linear with the number of threads. It is not surprising since the higher the number of threads is, the more data has to go through the network. However, the slope is higher for MVAPICH 2 since it suffers from contention caused by N competing receiver threads and sends all packets serialized through the network. On the other hand, `pioman`-enabled *NewMadeleine* has no contention for N receiver threads, and is able to aggregate [24] messages from multiple threads into a single packet sent on the wire.

The N-load benchmark evaluates communication progression when competing against computing threads. N computing threads on both sides are running, alongside a single-thread

1 MB pingpong. Results are depicted in Figure 12. We observe that pioman-enabled *NewMadeleine* and MVAPICH 2 have a constant median latency, while OpenMPI has a median latency linear with the number of computing threads for more than 16 threads (the machine has 32 cores). Moreover, *NewMadeleine* maximum latency is always lower than 20 ms, while OpenMPI maximum latency can be arbitrary high (graph is cropped).

These multithreaded benchmarks demonstrate that pioman progression engine is actually able to make communication efficiently progress when competing against other threads performing communications or computations, even with heavy load and oversubscribed cores. It exhibits better progression properties than state-of-the-art MPI implementations.

VI. Conclusion

With the dramatic increase in the number of cores per node in clusters, communication libraries have to deal with multithreading, and may exploit cores to make communication progress. However, mixing threads and communication is not straightforward, and care must be taken to design a thread-aware communication library.

In this paper, we have presented pioman, a generic framework to be used by communication libraries, that brings seamless asynchronous progression of communication. We have proposed mechanisms that make communication progress on timer events, opportunistically on idle cores, and allows explicit polling. Implementation uses system threads and thus is composable with any runtime system used for multithreading. We have studied tasks concurrency and proposed two mechanisms that mitigate contention, based on locality and an original locking scheme. We have shown that pioman overhead is low, and that it makes actually communication progress in background, thus allowing computation and communication to overlap. We have shown that it handles multithreaded load and does not collapse with massive number of threads.

In future works, we plan to modify MPI applications to actually take benefit from multithreading and to overlap computation and communications on both sides.We would like to extend pioman reach by porting a full featured MPI implementation on top of it, and study progression of collective operations. Finally, we are working on porting it to the Intel Xeon Phi.

References

[1] F. Trahay, É. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," in *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*. Rome, Italy: IEEE Computer Society Press, May 2009.

[2] É. Brunet, F. Trahay, and A. Denis, "A Multicore-enabled Multirail Communication Engine," in *Proceedings of the IEEE International Conference on Cluster Computing*. Tsukuba, Japan: IEEE Computer Society Press, Sep. 2008, pp. 316–321, poster Session.

[3] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: Multithreaded MPI for Many-core Environments," in *ACM International Conference on Supercomputing (ICS)*, Jun.

[4] J. Sancho, K. Barker, D. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM New York, NY, USA, 2006.

[5] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda, "Quantifying Performance Benefits of Overlap Using MPI-2 in a Seismic Modeling Application," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 17–25.

[6] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein, "Prospects for truly asynchronous communication with pure MPI and hybrid MPI/OpenMP on current supercomputing platforms."

[7] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A Flexible High Performance MPI," in *The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[8] S. Sur, H. Jin, L. Chai, and D. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM New York, NY, USA, 2006, pp. 32–39.

[9] M. J. Rashti and A. Afsahi, "Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects," in *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*. IEEE, 2008, pp. 95–101.

[10] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein, "Asynchronous MPI for the masses," *CoRR*, vol. abs/1302.4280, 2013.

[11] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing-to thread or not to thread?" in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 213–222.

[12] F. Trahay, A. Denis, O. Aumage, and R. Namyst, "Improving reactivity and communication overlap in MPI using a generic I/O manager," in *EuroPVM/MPI*, ser. LNCS, vol. Recent Advances in Parallel Virtual Machine and Message Passing Interface, no. 4757. Springer, 2007, pp. 170–177.

[13] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *Proceedings of the IEEE International Conference on Cluster Computing*. New Orleans, LA: IEEE Computer Society Press, Sep. 2009.

[14] Mindcraft, "Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 Upgraded to the Linux 2.2.2 Kernel," Tech. Rep., 1999. [Online]. Available: http://www.mindcraft.com/whitepapers/nts4rhlinux.html

[15] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux.conf.au*. Perth, Australia: The University of Western Australia, January 2003.

[16] S. Dandamudi, "Reducing run queue contention in shared memory multiprocessors," *Computer*, vol. 30, no. 3, pp. 82–89, Mar 1997.

[17] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa, Italie, Feb. 2010.

[18] S. Moreaud and B. Goglin, "Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines," in *PDCS*, Cambridge, États-Unis, 2007.

[19] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 214–222.

[20] P. E. Mckenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, Oct. 1998, pp. 509–518.

[21] A. Shet, P. Sadayappan, D. Bernholdt, J. Nieplocha, and V. Tipparaju, "A framework for characterizing overlap of communication and computation in parallel applications," *Cluster Computing*, vol. 11, no. 1, pp. 75–90, 2008.

[22] D. K. Panda, "OSU Micro-Benchmark." [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[23] R. Thakur and W. Gropp, "Test suite for evaluating performance of multithreaded MPI communication," *Parallel Computing*, vol. 35, no. 12, pp. 608–617, 2009.

[24] É. Brunet, O. Aumage, and R. Namyst, "Dynamic optimization of communications over high speed networks," in *HPDC-15, The 15th IEEE International Symposium on High Performance Distributed Computing*, Paris, Jun. 2006, poster Session.